Java EE

Interface Web et serveurs d'application
Version du 21/09/2009

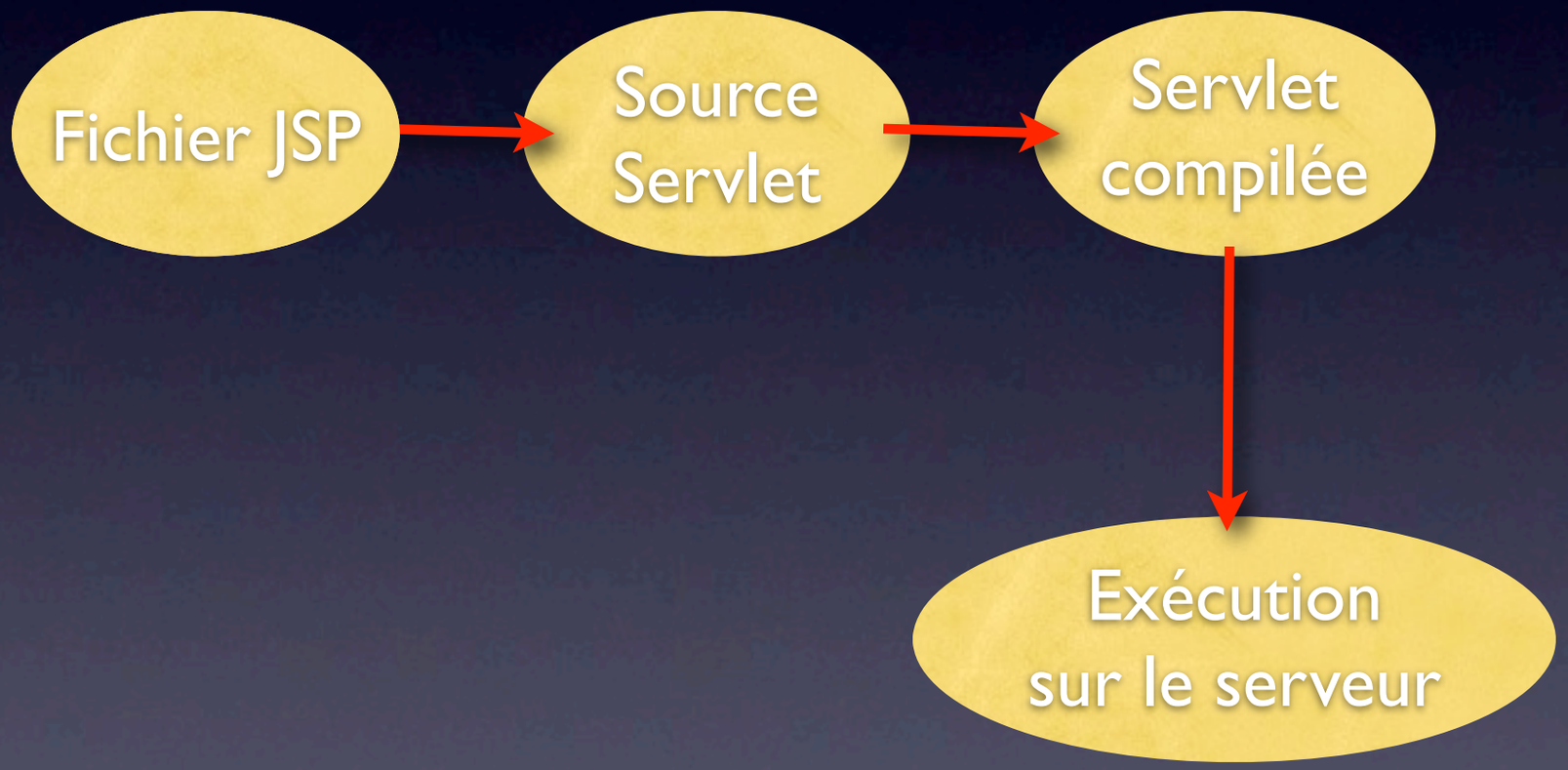
JSP : Java Server Pages

- Réponse de Sun à la vague des langages de Scripting
- A l'usage, très similaire à PHP
- Mais les sites "Full JSP" sont rares :
  - Découpage 3-tiers avec objets métiers
  - Utilisation de Frameworks (Struts...)

JSP et Servlets

- Historiquement, la notion de Servlet est plus ancienne
- Une page JSP n'est qu'une "greffe" sur un serveur de servlets :
  - Une page JSP est convertie en servlet avant exécution !

Cycle de vie


Fichier JSP
Source Servlet
Servlet compilée
Exécution sur le serveur

Balises spécifiques

- Exécution de code Java, sans affichage :

  <% int i=0; i=i+1; ... %>

  Sur une ou plusieurs lignes

- Affichage d'un élément Java :

  <%= i %>

  Un seul affichage, pas de ";"

- Exécution avec affichage :

```
<%
int i=0;
out.println("valeur de i :"+i);
%>
```

Exemple de page JSP

```
<html>
<head><title>Mixez Java et Balises !</title></head>
<body>
<p>Une balise avant du code</p>
<%
    int i=0;
    while (i < 5)
    {
%>
      <%= i%>
      <br>
<%
       ++i;
    }
%>
<p>Une balise après du code</p>
</body>
</html>
```

Affichage de i

Autre mode d'affichage

```
<html>
<head><title>Mixez Java et Balises !</title></head>
<body>
<p>Une balise avant du code</p>

<%
   int i=0;
   while (i < 5)
   {
     out.println(++i+"<br>");
   }
%>

<p>Une balise après du code</p>
</body>
</html>
```

Objets notoires

- Ce sont des instances de classes représentant divers outils au sein d'une page JSP
- Ces instances sont fournies par défaut, sans intervention de l'utilisateur
- Permet de gérer les E/S d'une page JSP

Exemples d'objets notoires

Objet notoire	Rôle
request	Entrées de la page (paramètres, cookies...)
response	Sortie de la page (cookie, flux texte ou binaire...)
out	Flux de sortie texte
session	Espace mémoire lié à la session utilisateur
application	Espace mémoire commun

Paramètres d'entrée

- On utilise l'objet notoire request :

  String param=request.getParameter("param");

- Tous les paramètres sont des String

- On peut ensuite les convertir avec divers class wrappers

String age_s=request.getParameter("age");

int age=Integer.parseInt(age_s);

Attention à la levée d'exception !

Le paramètre d'entrée

- On reçoit une chaîne de caractères, puisque l'URL est une chaîne
  - Ex : toto.jsp?nom=Bob
- Si la chaîne contient "", le paramètre était vide
  - Ex : toto.jsp?nom=
  - Si getParameter() retourne *null*, le paramètre était absent
    - ex : toto.jsp

Persistance d'informations

- On utilise la notion de session :
  - Espace mémoire lié à un utilisateur
  - Persistant tout au long de son passage sur le site
  - Se termine avec un Timeout d'activité

Liaison avec la session

- On utilise l'objet notoire "session"
  - Chaque élément dans la session est représentée par une chaîne de caractères
- Stockage dans la session :

  session.setAttribute("client",monClient);
- Récupération depuis la session :

  Client c=(Client)session.getAttribute("client");

Directives de page

- Chaque page JSP doit commencer par une balise permettant de la paramétrer

<%@page option1 option2... %>

- Il est possible de répéter cette balise pour multiplier les paramètres

Options de directives de page

- language=”java”
  - La plupart du temps, Java est le langage utilisé
- errorPage=”url”
  - Permet de déclarer une page à afficher en cas d’erreur
- import=”package1,package2”
  - Liste des packages utilisés par la page JSP

Exemple d'utilisation

```
<%@page import="java.util.*" errorPage="err.htm" %>
<html>
<body>
<%
ArrayList l=new ArrayList();
...
int age=Integer.parseInt(request.getParameter("age"));
...
%>
Bonjour, tu as <%=age%> ans !
</body>
</html>
```

Possible grâce à l'import

En cas d'erreur, *err.htm* sera affiché

Inclusion, redirection

- On peut fragmenter une page JSP via des “include” :

  <jsp:include page=”url” flush=”true” />

  - ☢ Page interprétée avant inclusion

- On peut forcer une redirection avec :

  <jsp:forward page=”url” />

- Mais on préfère souvent l’objet notoire :

  response.sendRedirect(“url”);

Liaison avec la partie métier

- Dans un schéma 3-tiers, le JSP n'est là que pour le côté "vue"
- Pour tous les traitements, il fait appel à des objets métiers :
  - instanciés sur mesure
  - ou stockés au préalable en session

Ex. de liaison métier

```
<%@page import=”metier.*” errorPage=”err.htm” %>
<html><body>
<%
Client c=new Client();
c.setNom(request.getParameter(“nom”));
c.setPrenom(request.getParameter(“prenom”));
session.setAttribute(“client”,c);
%>
Bonjour, <%=c.getNom()%> !
</body>
</html>
```

Ecriture de servlet

- Dans certains cas, il peut être utile d'écrire directement une servlet plutôt qu'un JSP :
  - Lorsqu'il n'y a aucun affichage à faire (ex : contrôleur)
  - Lorsqu'on veut distinguer un traitement GET d'un POST
  - Pour générer un flux binaire
  - Cas de la servlet d'initialisation

Servlet d'initialisation

- Il peut être utile de lancer une séquence d'initialisation en même temps que le serveur d'application

- Ecrire une servlet avec une méthode init()
- Insérer dans web.xml, dans la balise <servlet>, un :

```
<load-on-startup> 1 </load-on-startup>
```

Utilisation des beans

- Permet de faciliter la liaison entre la partie JSP et les objets métiers
  - Gère la persistance des instances (session..)
  - Automatise les transferts de paramètres via des get/set automatisés
- Ne peut s'appliquer qu'à certains cas
  - Ex : formulaires complexes
  - Sinon, on utilisera plutôt des frameworks (Struts...)

Qu'est ce qu'un bean ?

- C'est avant tout une classe Java
- Avec deux particularités :
  - Au moins un constructeur sans paramètre
  - des get/set pour les attributs accessibles depuis l'extérieur

Exemple de Bean

```
public class Client
{
  private String nom;
  private String prenom;
  public Client() { ... }
  public String getNom() { return this.nom; }
  public String getPrenom() { return this.prenom; }
  public void setNom(String nom) { this.nom=nom; }
  public void setPrenom(String prenom)
  { this.prenom=prenom; }
}
```

Gestion de l'instanciation d'un bean

- Pour certaines classes se pose la question du : "quand l'instancier"
  - Ex : un panier d'achat
  - Sur un site Web, on ne maitrise pas forcément le "point d'entrée
- Le système de beans propose une alternative au new(...) classique (proche du singleton)

Syntaxe d'instanciation

<jsp:useBean class="..." id="..." scope="..." />

- Paramètres :
- class : nom de la classe à instancier (y compris le nom du package)
- id : nom symbolisant l'instance (dans la page, et éventuellement dans la session)
- scope : type de persistance

Persistance d'un bean

- scope="request" : le bean sera réinstancié à chaque chargement de la page
- scope="page" : le bean ne sera instancié qu'une fois sur cette page
- scope="session" : le bean est commun à toutes les pages pour une session utilisateur
- scope="application" : le bean est commun à tous les utilisateurs

Exemple d'une gestion panier

- La notion de bean permet de gérer un panier d'une manière très simple :
  - il faut recopier en chaque début de page la ligne :

```
<jsp:useBean class="fr.ecoms.Panier" id="panier"
scope="session" />
```

- Sur la page, on dispose ensuite d'une instance nommée "panier"

```
<% panier.ajout(unProduit); %>
```

Affectation de valeurs à un bean

- Principe : appel implicite aux setters du bean
- On utilise la balise setProperty, avec plusieurs modes d'utilisation.

- Commençons par le "moins" utile :

```
<jsp:setProperty name="news"
property="num" value="12" />
```

- Cette ligne est équivalente à :

```
<% news.setNum(12); %>
```

Récupération de paramètre & affectation

- Sans “value”, la valeur va être récupérée dans les paramètres d’entrée de la page :

```
<jsp:setProperty name=”news”
property=”num” />
```

- Cette ligne est équivalente à :

```
<% news.setNum(
      Integer.parseInt(
         request.getParameter(“num”)); %>
```

Même la conversion est automatisée !

Récupération de tous les paramètres

- Avec property=”*”, une liaison sera faite entre tous les paramètres et les setters correspondants du bean

```
<jsp:setProperty name=”news”
property=”*” />
```

- Utile en récupération par ex. d’un grand formulaire d’entrée
  - Attention à la sécurité !

Utilisation de la JSTL

- Spécification de tags additionnels définis par Sun
- Destinés à faciliter l'écriture de pages JSP
  - En limitant au maximum l'utilisation de code Java
- Mais pas d'implémentation officielle
  - On utilise des taglibs d'éditeurs tiers

Syntaxe : les EL

- Expression Language
- Permet d'écrire des accès aux données
- Syntaxe de base : ${....}
- Accès à un attribut de bean :
  - <%=bean.getNom() %>
  - ${bean.nom}
  - ${bean["nom"]}

Librairie “core” : <c:... />

- Comme toutes les taglibs, on définit le préfixe :
- <%@taglib uri=”http://java.sun.com/jstl/core” prefix=”c” %>
- <c:out /> permet d’afficher une expression
- En gérant d’éventuelles valeurs par défaut

  <c:out value=”$sessionScope[‘client’].nom” default=”Inconnu” />

Affectation : <c:set />

<c:set scope=”session” var=”client”
value=”${requestScope[‘nom’]}” />

<c:set target=”$session[‘client’]”
property=”nom” value=”...” />

Tests

```
<c:if test=”{$empty param[‘ref’]}”>
...
</c:if>
```

```
<c:choose>
<c:when test=”{value==1}”> ..... </c:when>
...
<c:otherwise> ....</c:otherwise>
</c:choose>
```

Le Framework Struts

- Principes d'un framework :
  - Lier et structurer différents éléments en une architecture "fixée"
- Dans le cas de Struts :
  - Fixer un comportement MVC
  - Automatiser les allez/retour entre Vue et Contrôleur
  - http://struts.apache.org

Rappels sur MVC

- *Modèle* : les objets métiers, contenant les comportements et traitements
- *Vue* : L'interface utilisateur (page JSP)
- *Contrôleur* : élément liant Vue et Modèle :
  - récupère et contrôle les params d'entrée
  - gère les objets métiers (appel, persistance)
  - définit les redirections

Struts et MVC

- Struts va prendre en charge les échanges entre vue et contrôleur
  - En insérant dans la vue des balises particulières de récupération/traitement
  - En fournissant un “cadre” d’écriture du contrôleur

Structure d'une page Struts basique

- 3 éléments principaux :
  - Une page JSP avec les *taglibs* Struts
  - Une classe Action (le contrôleur)
  - Un fichier de mapping (configuration XML)

La vue sous Struts

- C'est une page JSP
- Utilisant des taglibs (balises personnalisées)

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h2><s:property value="message" /></h2>
    </body>
</html>
```

Insertion d'un texte par Struts

L'action

- C'est un bean représentant à la fois les données de la vue, et les actions du contrôleur

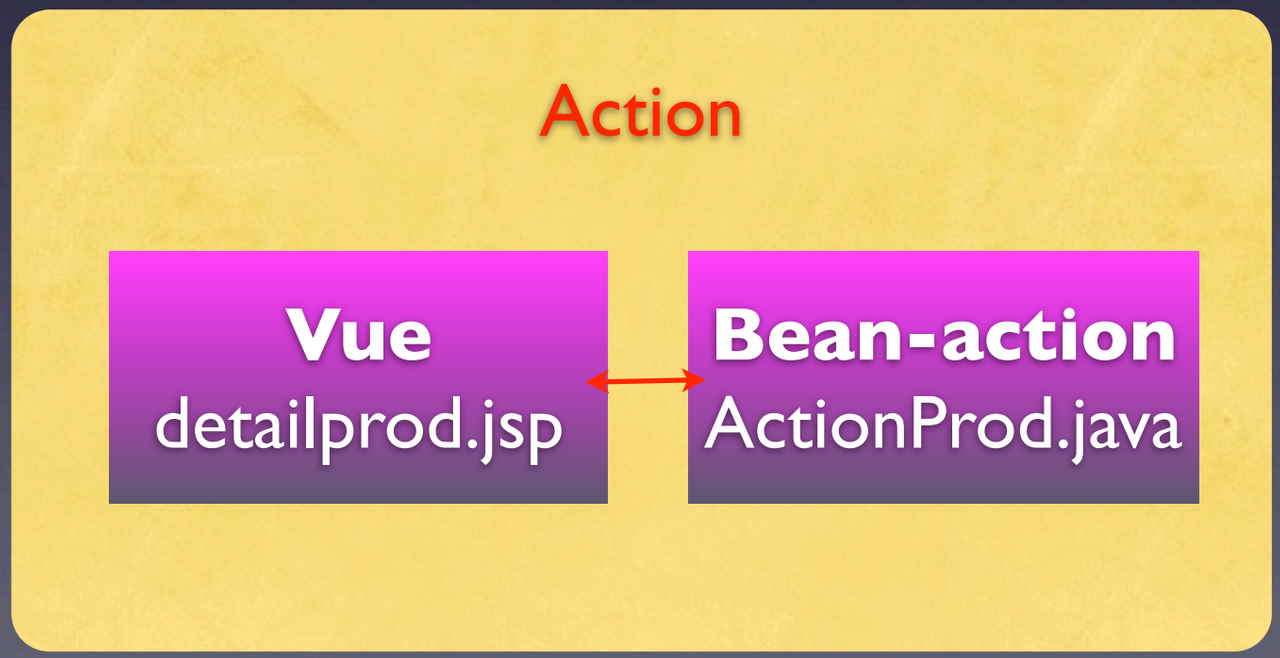
```
package tutorial;
import com.opensymphony.xwork2.ActionSupport;
public class HelloWorld extends ActionSupport {

    public static final String MESSAGE = "Hello, World !";

    public String execute() throws Exception {
        setMessage(MESSAGE);
        return SUCCESS;
    }

    private String message;
    public void setMessage(String message){   }
    public String getMessage() {  }
}
```

Méthode appelée par Struts pour afficher la vue

Le bean représente les données de la page, pas du métier !

Liaison entre vue et bean-action

- Le bean est là pour représenter les données dynamiques de la vue
  - Permet de travailler sur le bean (=classe, +facile à accéder) plutôt que sur la vue (=page JSP, d'accès plus complexe)



Le fichier de mapping

- Le fichier struts.xml décrit les liaisons entre vues et contrôleurs
- Ainsi que les redirections

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <package name="tutorial" extends="struts-default">
        <action name="HelloWorld" class="tutorial.HelloWorld">
            <result>/helloworld.jsp</result>
        </action>
        <!-- Add your actions here -->
    </package>
</struts>
```

Appel de la page

- On lance l'URL :
- http://localhost/essai/HelloWorld.action

- Le .action indique qu'on passe par Struts :
  - Exécution de l'action HelloWorld.java
  - puis affichage de la vue helloworld.jsp

Enchaînement contrôleur/vue

http://...../HelloWorld.action

HelloWorld.java

```
execute()
{
...
return SUCCESS;
}
```

helloworld.jsp

<html>....

Le bean-action n'est pas un bean métier !

- Comme un bean métier, le bean action contient des données
  - Mais des données représentant la page, pas le métier !
- La liaison entre partie métier et le bean-action se fait dans la méthode *execute()*

Exemple d'appel métier

```
public class ActionProd extends ActionSupport
{
  private String nom;
  public void execute()
  {
    Client c=Client.getClient(12);
    this.nom=c.getNom();
    return SUCCESS;
  }
}
```

Donnée qui sera affichée dans la page

Appels métiers

Les données métiers sont transférées dans le bean action

Définition d'un formulaire

- On utilise des balises Struts :

```
<s:form action="Logon.action">
  <s:textfield label="User Name" name="username"/>
  <s:password label="Password" name="password" />
  <s:submit/>
</s:form>
```

  - Le formulaire appelle une action
  - Chaque champ est rempli avec des attributs du contrôleur lié

Contrôle du formulaire

```
package tutorial;
import com.opensymphony.xwork2.ActionSupport;
public class Logon extends ActionSupport {

    public String execute() throws Exception {

        if (isInvalid(getUsername())) return INPUT;
        if (isInvalid(getPassword())) return INPUT;
        return SUCCESS;
    }

    private boolean isInvalid(String value) {
        return (value == null || value.length() == 0);
    }

    private String username;
    public String getUsername() {  }
    public void setUsername(String username) {   }

    private String password;
    public String getPassword() {   }
    public void setPassword(String password) {  }
}
```

Retour à la page de saisie (avec champs préremplis)

Mapping du formulaire

- On insère la description de l'action dans le fichier de mapping

```
<action name="Logon" class="tutorial.Logon">
  <result name="input">/Logon.jsp</result>
  <result type="redirect-action">Menu</result>
</action>
```

- La classe Logon contient à la fois les données du formulaire et les tests
- Logon.jsp est la vue correspondante
- En cas de succès, on redirige vers l'action *Menu*

Redirections

- S'il n'y a qu'un <*result*>, ce résultat est pris par défaut lorsqu'on renvoie la constante SUCCESS
- Si l'on veut faire des redirections conditionnelles, il faut les nommer avec des chaînes de caractères

```
<result name="”suite”">/suite.jsp</result>

public String execute()
{
   ... return “suite”;
}
```

Redirections globales

- Il est également possible de définir des <*result*> en dehors d'une action, pour pouvoir les utiliser à tout moment :

```
<package ....>
  <global-results>
    <result.... >
    ...
  </global-results>
  ...
</package>
```

Externalisation des messages

- Struts permet d'externaliser tous les messages texte dans un fichier externe
- Intérêts :
  - Pouvoir corriger les messages sans avoir à toucher au programme
  - Pouvoir multiplier les fichiers externes de messages pour fournir des traductions

Fichier externe de messages

- C'est un simple fichier *.properties*
  - package.properties en config "générale"
  - NomClasse.properties pour une classe en particulier
- Qui peut contenir du code HTML

```
HelloWorld.message=Hello, World !
welcome.msg=Bienvenue &agrave; tous !
titre.default=<b>Bonjour !</b>
```

Internationalisation (i18n)

- Principe : on multiplie les fichiers contenant les messages
  - Un fichier par langue, suffixé
    - Ex : package_fr.properties
  - On passe d'une langue à l'autre en passant le paramètre *request_locale=XX* sur une page quelconque
    - Ex : Menu.action?request_locale=fr

Insertion d'un message

- Dans une vue JSP :

```
<s:text name="HelloWorld.message" />
<s:property value="%{getText
('HelloWorld.message')}" />
```

- Dans la classe d'action :

```
getText("HelloWorld.message");
```

Insertion dans une balise quelconque

Validation paramétrée du formulaire

- On utilise un fichier XML de description des règles de validation
- Nom du fichier : *nomdelaclasse-validation.xml*

```
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>
    <field name="username">
        <field-validator type="requiredstring">
            <message>Username is required</message>
        </field-validator>
    </field>
    <field name="password">
        <field-validator type="requiredstring">
            <message>Password is required</message>
        </field-validator>
    </field>
</validators>
```

Outils de validation

- <field-validator type=”XX”> peut prendre différentes valeurs :
  - *required* : champ obligatoire
  - *requiredstring* : chaîne obligatoire
  - *int* : nombre obligatoire
  - *email* : email obligatoire
  - ...

Paramètres de validation

- La plupart des outils de validation peuvent prendre des paramètres :

```
<field-validator type="int">
    <param name="min">1</param>
    <param name="max">100</param>
    <message key="invalid.count">
        Value must be between ${min} and${max}
    </message>
</field-validator>
```

Exécution de la validation

- Lors de chaque appel de l'action, la validation est appelée au préalable
  - Les messages sont affichés dans la vue par :

<s:actionmessage />

- Problème : la validation est donc appelée lors du premier appel de l'action
- Parade : paramétrer l'appel de l'action

Appeler une action en mode “input”

- Appel initial de l’action : on utilise le suffixe “_input” pour neutraliser la validation :

```
<a href="<s:url action="Logon_input"/>">Se connecter</a>
```

- Dans le fichier struts.xml, on récupère le suffixe :

```
<action name="Logon_*" method="{1}" class="tutorial.Logon">
```

- Du coup, un appel à *Logon_input.action* va appeler *input()* au lieu de *execute()*, et ne lancera pas de validation

Actions multiples

- D'une manière générale, il est possible d'exploiter une même classe action pour gérer des actions multiples
  - On précise simplement la méthode à appeler

- En fait, *execute()* est appelé par défaut dans le cas d'une seule action

```
<action name="list" method=”execute" class="tutorial.Logon"> ...
<action name="save" method=”sauve" class="tutorial.Logon"> ...
<action name="delete" method=”efface" class="tutorial.Logon"> ...
```

Validation avec messages externalisés

- Dans le fichier des messages :

  required.msg=${getText(fieldName)} doit être saisi !

- Dans Logon-validation.xml :

  <message key=”required.msg” />

Passerelle Java/PHP

- Objectif : appeler des classes Java depuis une interface PHP
- Eventuellement dans l'autre sens

- Plusieurs implémentations disponibles
  - "L'officielle" tendant à rester expérimentale (obsolète ?)
  - http://php-java-bridge.sf.net

Instanciation d'une classe

- On utilise la classe "Java" :

```
$liste=new Java("java.util.ArrayList");

$c=new Client
("metier.Client","bob","l'éponge");
```

Appels de méthodes

- Les objets récupérés via new Java(..) sont utilisables comme des objets PHP :

```
$c->genereFacture();
```

- Les paramètres d'entrée sont automatiquement convertis :

```
$liste->add("chaine");
```

Parcours de listes

- Tous les objets de type liste, tableau... sont parcourables via un *foreach(..)* PHP :

```
foreach($liste as $valeur)
{
   ...
}
```